MEMOIZAÇÃO OU ITERAÇÃO?

- Nos slides anteriores utilizados o problema do escalonamento de tarefas ponderadas para demonstrar os princípios de Programação Dinâmica.
- Também apresentamos duas perspectivas para a solução de problemas: recursivamente ou iterativamente.
- Na abordagem recursiva começamos com um algoritmo em tempo exponencial e depois utilizamos memoização para obter tempo polinomial.
- Para compreendermos o processo de Programação Dinâmica, vamos propor uma reformulação completamente equivalente do algoritmo de memorização.
- Esta nova formulação é a que melhor captura a essência da Programação Dinâmica e servirá para o desenvolvimento de algoritmos futuros.

O ALGORITMO:

- A chave para o algoritmo eficiente é o vetor M.
- Ele representa a noção de que estamos usando um valor de soluções ótimas de subproblemas em intervalos $\{1,2,3, \ldots, j\}$ para cada j.
- E utiliza **A.20**

$$\mathbf{OPT}(j) = \begin{cases} \mathbf{0} & , se\ j = 0 \\ max\left(v_j + \mathbf{OPT}(p(j)), \mathbf{OPT}(j-1)\right) & , se\ j > 0 \end{cases}$$

- para definir o valor de $M[j]$ baseado em valores que antecedem j no vetor.
- Uma vez que temos o vetor M, o problema está resolvido, pois $M[n]$ contém o valor da solução ótima para a instância completa do problema.
- Depois utilizamos Find-Solution para obter o conjunto com todos os intervalos que compõem a solução ótima.
- O ponto que precisamos observar é que podemos diretamente computar as entradas em M via um algoritmo iterativo, ao invés de uma recursão com memorização.

- Começamos com $M[0] = 0$ e seguimos incrementando j.
- Cada vez que precisarmos computar $M[j]$, simplesmente utilizamos **A.20.**
- Portanto, temos o seguinte algoritmo Iterative-Compute-Opt :

```
Iterative-Compute-Opt
   M[0] = 0
   For j = 1, 2, . . . , n
      M[j] = max(v_j + M[p(j)], M[j − 1])
   Endfor
```

- Lembrando apenas que assumimos que:
  - já ordenamos as requisições pelo tempo de finalização
  - Computamos os valores de $p(j)$ para cada j.

PUCRS ESCOLA POLITÉCNICA

Programação Dinâmica:

Princípios da Programação Dinâmica

```
Iterative-Compute-Opt
  M[0] = 0
  For j = 1, 2, . . . , n
    M[j] = max(v_j + M[p(j)], M[j − 1])
  Endfor
```

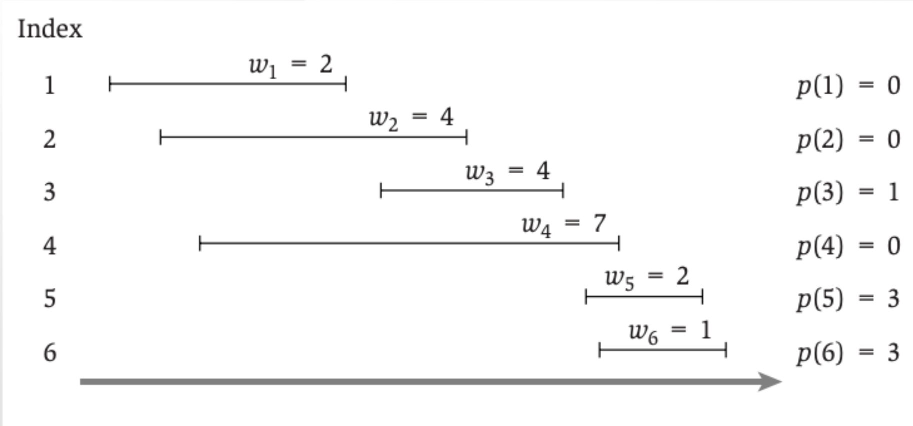
ANALISANDO O ALGORITMO:

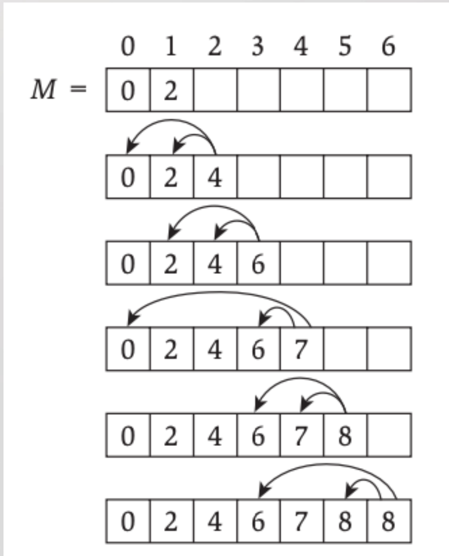
- Por analogia exata com a prova de **A.22**

A.22: Compute-Opt(j) **computa corretamente** $\mathbf{OPT}(j)$ **para cada** $\boldsymbol{j = 1, 2, \ldots, n}$.

- Podemos demonstrar por indução em j que este algoritmo escreve $\mathrm{OPT}(j)$ numa entrada $M[j]$.
- **A.20** nos entrega o passo indutivo.
- Adicionalmente, como antes, podemos passar o vetor preenchido M para Find-Solution para obter as transações que representam o conjunto ótimo.
- Por fim, o tempo de execução de Iterative-Compute-Opt é claramante $O(n)$, pois ele executa por n iterações e gasta $O(1)$ em cada uma.

- Um exemplo da execução de Iterative-Compute-Opt é apresentado abaixo.
- O algoritmo preenche uma entrada no vetor M comparando o valor de $v_j + M[p(j)]$ com $M[j-1]$.





IMPLEMENTAÇÃO E TEMPO DE EXECUÇÃO:

Sort jobs by finish time and renumber so that $f_1 \leq f_2 \leq \ldots \leq f_n$.

Compute $p[1], p[2], \ldots, p[n]$.

$M[0] \leftarrow 0.$

previously computed values

For $j = 1$ **to** n

$\quad M[j] \leftarrow \max \{ M[j-1],\ w_j + M[p[j]] \}.$

Utilizamos merge-sort para ordenar - $O(n \log n)$

Utilizamos busca binária - $O(n \log n)$

Iterative-Compute-Opt **-** $O(n)$

Find-Solution(j)

If $(j = 0)$

$\quad$ **Return** $\varnothing$.

Else if $(w_j + M[p[j]] > M[j-1])$

$\quad$ **Return** $\{ j \} \cup$ Find-Solution($p[j]$).

Else

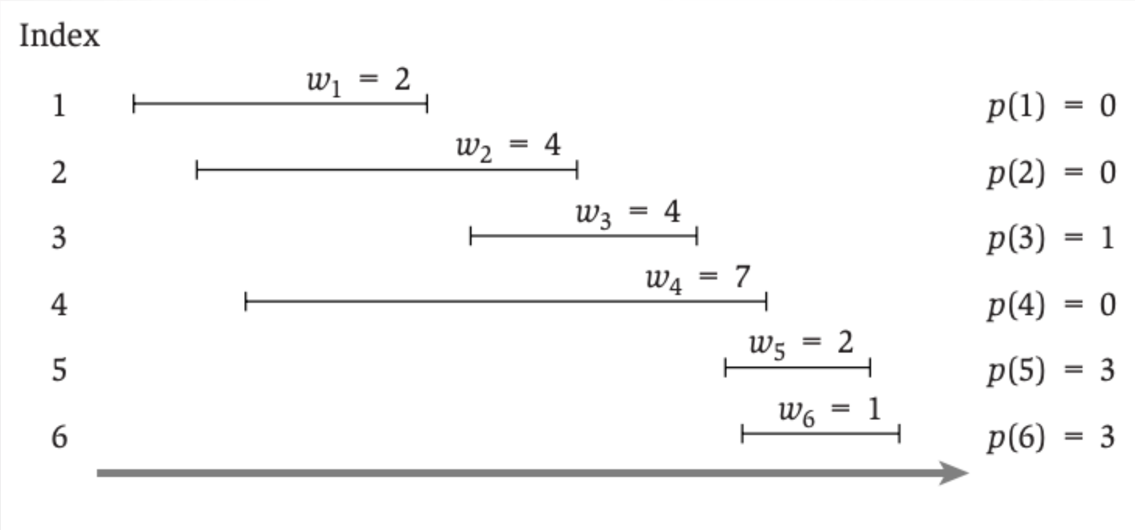
$\quad$ **Return** Find-Solution($j-1$).

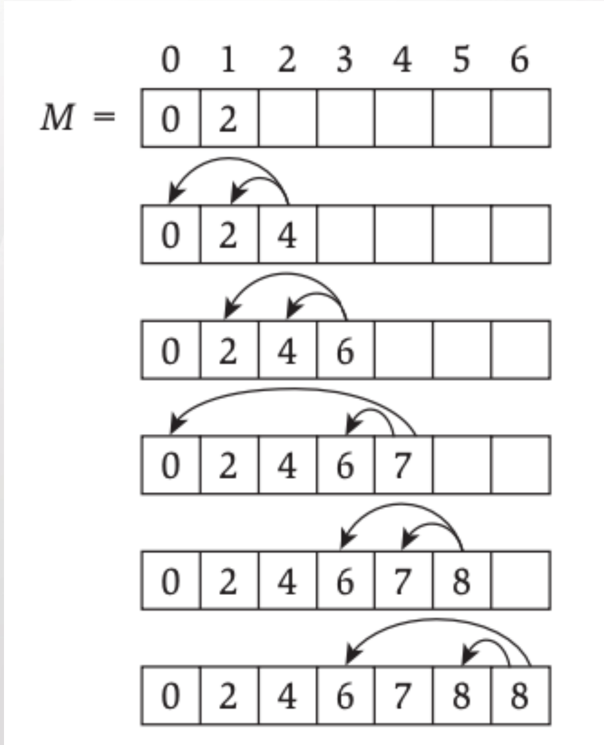
Find-Solution **-** $O(n)$

- **Tempo total do algoritmo de busca de valores (Iterative-Compute-Opt) e busca de soluções (Find-Solution) é $O(n \log n)$.**

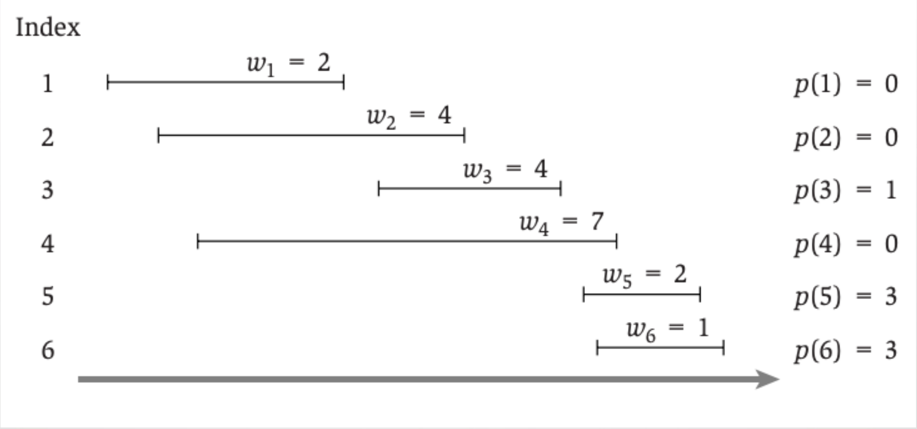
Programação Dinâmica:
Princípios da Programação Dinâmica

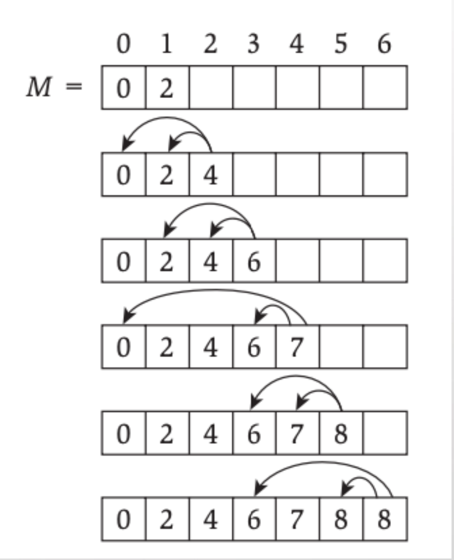
IMPLEMENTAÇÃO E TEMPO DE EXECUÇÃO:





IMPLEMENTAÇÃO E TEMPO DE EXECUÇÃO:

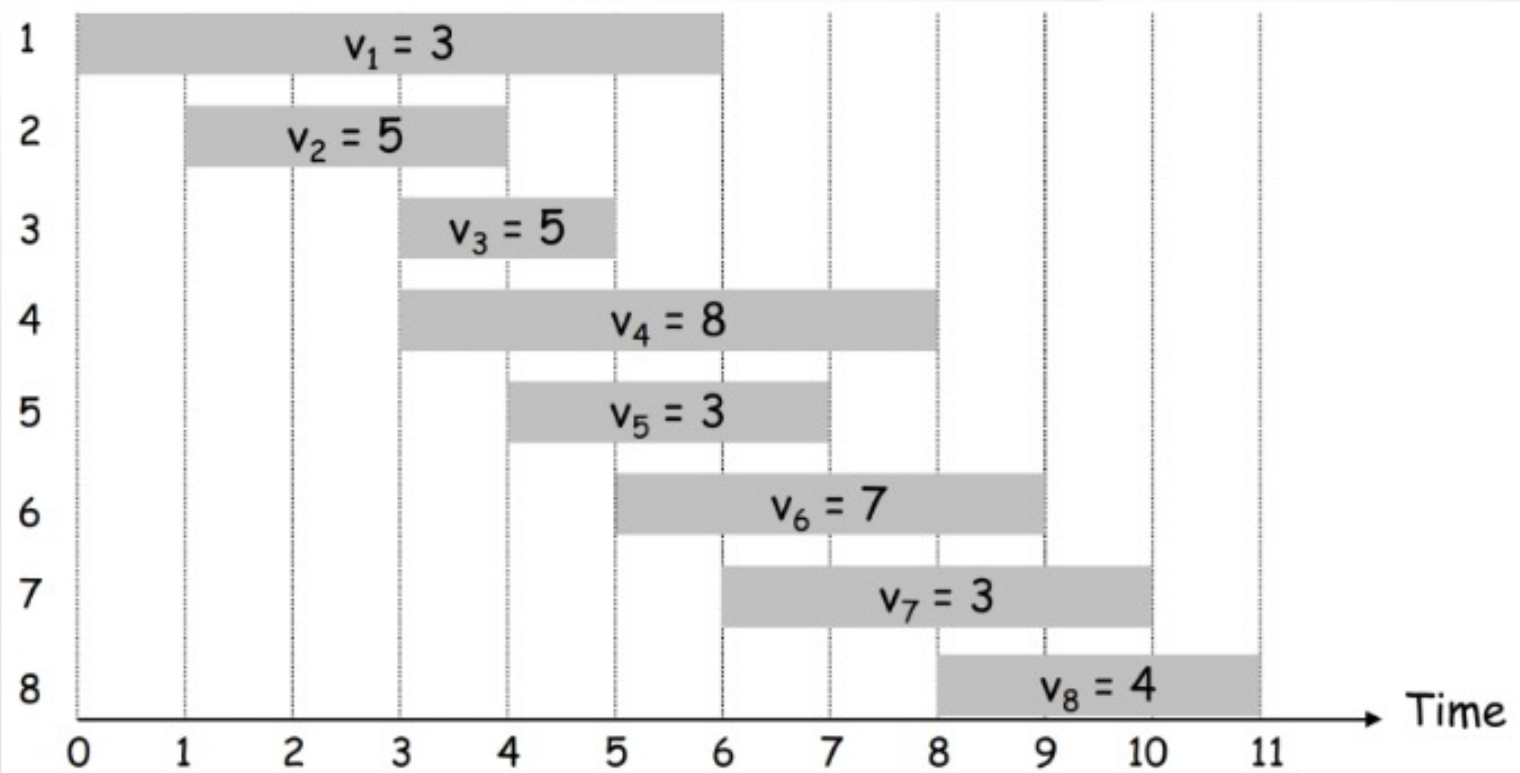




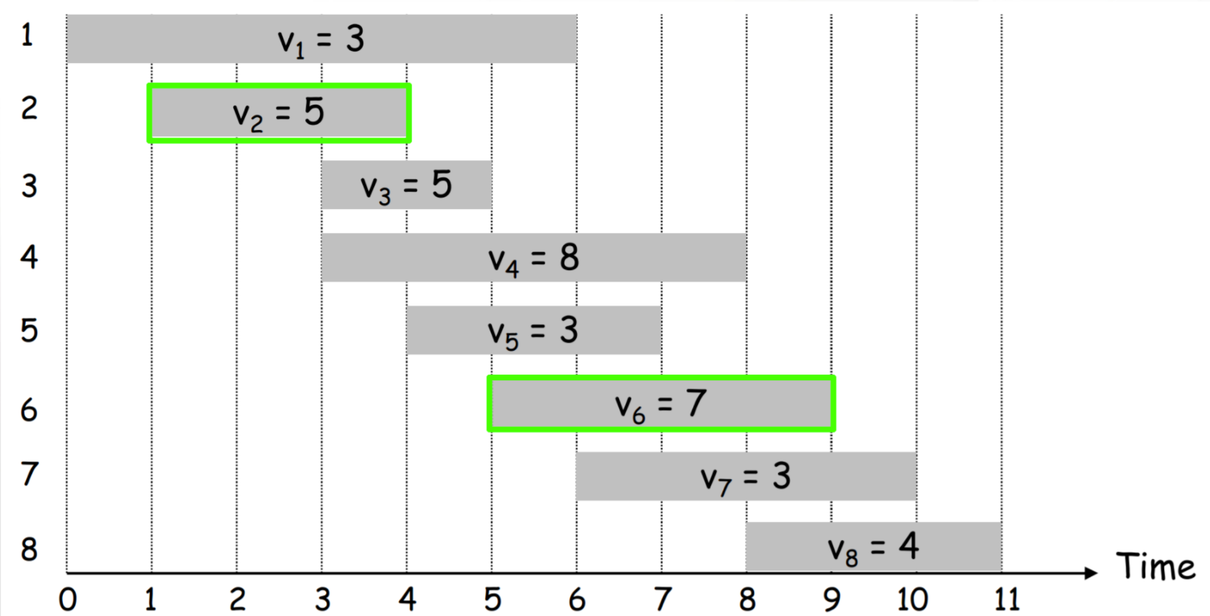

```
Vetor de memoizacao: [0, 2, 4, 6, 7, 8, 8]
Valor máximo considerando o conjunto ótimo de solicitacoes = 8

Requisicao incluida na solucao final:
Requisicao 1:  Tempo (0-4) Valor=2
Requisicao 3:  Tempo (5-7) Valor=4
Requisicao 5:  Tempo (8-11) Valor=2
```

IMPLEMENTAÇÃO E TEMPO DE EXECUÇÃO:



IMPLEMENTAÇÃO E TEMPO DE EXECUÇÃO:




```
Vetor de memoizacao: [0, 5, 5, 5, 8, 8, 12, 12, 12]
Valor máximo considerando o conjunto ótimo de solicitacoes = 12

Requisicao incluida na solucao final:
Requisicao 2:  Tempo (1-4) Valor=5
Requisicao 6:  Tempo (5-9) Valor=7
```

CARACTERÍSTICAS BÁSICAS DA PROGRAMAÇÃO DINÂMICA

- Apresentamos duas perspectivas para a solução de problemas: recursivamente ou iterativamente.
- A abordagem iterativa é a mais utilizada na prática, devido a sua simplicidade frente ao processo recursivo.
- Porém, ambas abordagens são equivalentes conforme demonstrado.
- Para desenvolver um algoritmo baseado em programação dinâmica, precisamos de uma coleção de subproblemas derivados do problema original que satisfaça algumas propriedades básicas:
    1. Existe apenas um número polinomial de subproblemas.
    2. A solução do problema original pode ser facilmente computada das soluções dos subproblemas. (Por exemplo, o problema original pode na verdade ser um dos subproblemas.)
    3. Existe uma ordenação natural de "menor" para "maior" (no sentido do tamanho das recorrências) nos subproblemas, juntamente com uma recorrência de fácil computação como em **(A.20 e A.21)** que permite a determinação de uma solução para o subproblema de soluções para um número menor de subproblemas.
- Algumas vezes é mais fácil começar o processo de desenho do algoritmo pela formulação de subproblemas que parecem naturais e intuitivos, e então definir uma recorrência que combina esses subproblemas;
- Outras vezes, podemos começar definindo uma recorrência pela análise da estrutura da nossa solução ótima, e então determinar que subproblemas serão necessários para desenrolar a recursão.